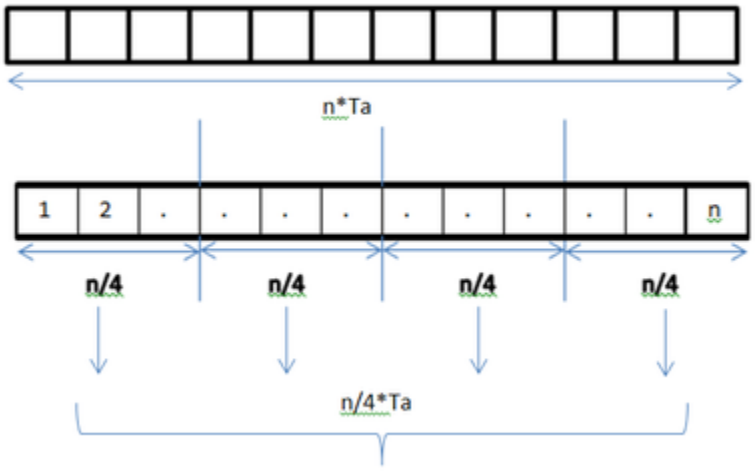
Unit-4

Data parallelism is parallelization across multiple processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel. It contrasts to task parallelism as another form of parallelism.

A data parallel job on an array of *n* elements can be divided equally among all the processors. Let us assume we want to sum all the elements of the given array and the time for a single addition operation is Ta time units. In the case of sequential execution, the time taken by the process will be *n*×Ta time units as it sums up all the elements of an array. On the other hand, if we execute this job as a data parallel job on 4 processors the time taken would reduce to (*n*/4)×Ta + merging overhead time units. Parallel execution results in a speedup of 4 over sequential execution. One important thing to note is that the locality of data references plays an important part in evaluating the performance of a data parallel programming model. Locality of data depends on the memory accesses performed by the program as well as the size of the cache.



The program expressed in pseudocode below—which applies some arbitrary operation, foo, on every element in the array d—illustrates data parallelism.

```
if CPU = "a" then
   lower_limit := 1
   upper_limit := round(d.length / 2)
else if CPU = "b" then
   lower_limit := round(d.length / 2) + 1
   upper_limit := d.length

for i from lower_limit to upper_limit by 1 do
   foo(d[i])
```

In an SPMD system executed on 2 processor system, both CPUs will execute the code.

Data parallelism emphasizes the distributed (parallel) nature of the data, as opposed to the processing (task parallelism). Most real programs fall somewhere on a continuum between task parallelism and data parallelism.

In computing, **SPMD** (**single program, multiple data**) is a technique employed to achieve parallelism;. SPMD is the most common style of parallel programming. It is also a prerequisite for research concepts such as active messages and distributed shared memory.

Task parallelism (also known as **function parallelism** and **control parallelism**) is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing tasks—concurrently performed by processes or threads—across different processors. In contrast to data parallelism which involves running the same task on different components of data, task parallelism is distinguished by running many different tasks at the same time on the same data.[1] A common type of task parallelism is pipelining which consists of moving a single set of data through a series of separate tasks where each task can execute independently of the others.

In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work, but this is not a requirement. Communication usually takes place by passing data from one thread to the next as part of a workflow.[2]

As a simple example, if a system is running code on a 2-processor system (CPUs "a" & "b") in a parallel environment and we wish to do tasks "A" and "B", it is possible to tell CPU "a" to do task "A" and CPU "b" to do task "B" simultaneously, thereby reducing the run time of the execution. The tasks can be assigned using conditional statements as described below.

The pseudocode below illustrates task parallelism:

```
program:
...
if CPU="a" then
  do task "A"
else if CPU="b" then
  do task "B"
end if
...
end program
```

The goal of the program is to do some net total task ("A+B"). If we write the code as above and launch it on a 2-processor system, then the runtime environment will execute it as follows.

- In an SPMD (single program, multiple data) system, both CPUs will execute the code.
- In a parallel environment, both will have access to the same data.
- The "if" clause differentiates between the CPUs. CPU "a" will read true on the "if" and CPU "b" will read true on the "else if", thus having their own task.
- Now, both CPU's execute separate code blocks simultaneously, performing different tasks simultaneously.

Code executed by CPU "a":

```
program:
```

```
...
do task "A"
...
end program
```

Code executed by CPU "b":

```
program:
...
do task "B"
...
end program
```

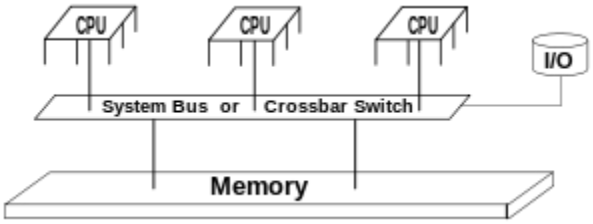
This concept can now be generalized to any number of processors.

Data parallelism vs. task parallelism[edit]

Data parallelism	Task parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed.
Designed for optimum load balance on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.

shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.

Using memory for communication inside a single program, e.g. among its multiple threads, is also referred to as shared memory.



An illustration of a shared memory system of three processors.

In computer hardware, *shared memory* refers to a (typically large) block of random access memory (RAM) that can be accessed by several different central processing units (CPUs) in a multiprocessor computer system.

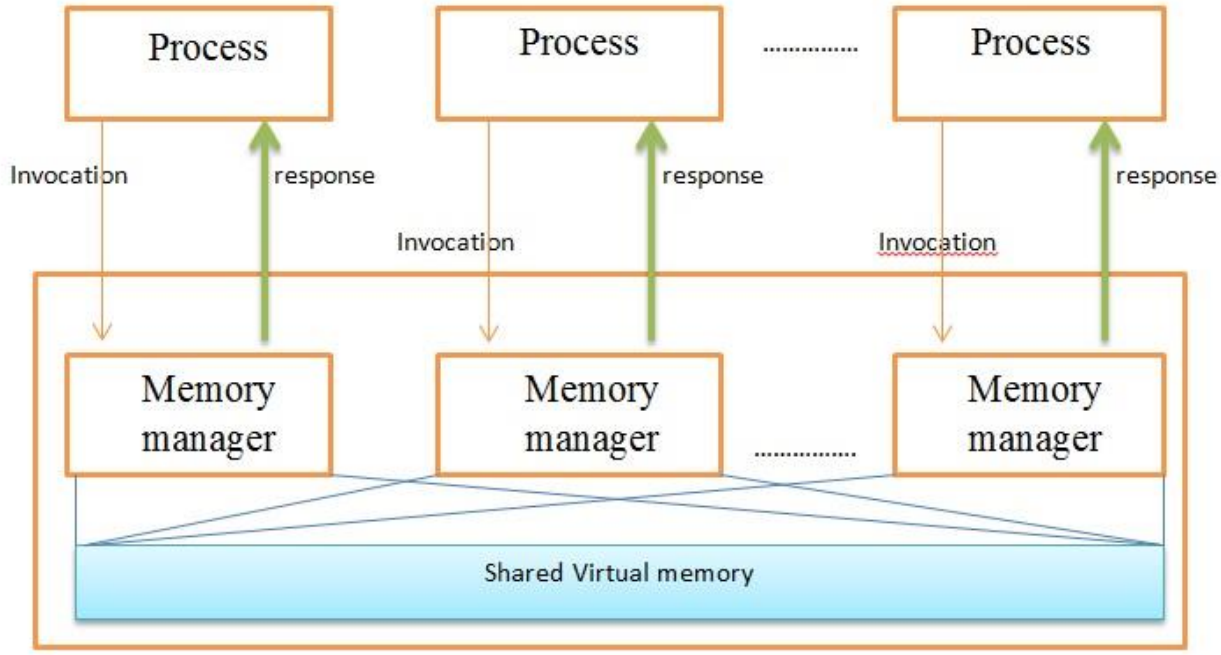
Shared memory systems may use:

- uniform memory access (UMA): all the processors share the physical memory uniformly;
- non-uniform memory access (NUMA): memory access time depends on the memory location relative to a processor;
- cache-only memory architecture (COMA): the local memories for the processors at each node is used as cache instead of as actual main memory.

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. The issue with shared memory systems is that many CPUs need fast access to memory and will likely cache memory, which has two complications:

- access time degradation: when several processors try to access the same memory location it causes contention. Trying to access nearby memory locations may cause false sharing. Shared memory computers cannot scale very well. Most of them have ten or fewer processors;
- lack of data coherence: whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data. Such cache coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processors. On the other hand, they can sometimes become overloaded and become a bottleneck to performance.

Shared memory architecture may involve separating memory into shared parts distributed amongst nodes and main memory; or distributing all memory between nodes. A coherence protocol, chosen in accordance with a consistency model, maintains memory coherence.



Distributed shared memory

Advantages

- Scales well with a large number of nodes
- Message passing is hidden
- Can handle complex and large databases without replication or sending the data to processes
- Generally cheaper than using a multiprocessor system
- Provides large virtual memory space
- Programs are more portable due to common programming interfaces
- Shield programmers from sending or receiving primitives

Disadvantages

- Generally slower to access than non-distributed shared memory
- Must provide additional protection against simultaneous accesses to shared data
- May incur a performance penalty
- Little programmer control over actual messages being generated
- Programmers need to understand consistency models, to write correct programs
- DSM implementations use asynchronous message-passing, and hence cannot be more efficient than message-passing implementations

message passing is a technique for invoking behavior (i.e., running a program) on a computer. The invoking program sends a message to a process (which may be an actor or object) and relies on that process and its supporting infrastructure to select and then run the code it selects. Message passing differs from

conventional programming where a process, subroutine, or function is directly invoked by name. Message passing is key to some models of concurrency and object-oriented programming.

Message passing is used ubiquitously in modern computer software. It is used as a way for the objects that make up a program to work with each other and as a means for objects and systems running on different computers (e.g., the Internet) to interact. Message passing may be implemented by various mechanisms, including channels.

Message passing is a technique for invoking behavior (i.e., running a program) on a computer. In contrast to the traditional technique of calling a program by name, message passing uses an object model to distinguish the general function from the specific implementations. The invoking program sends a message and relies on the object to select and execute the appropriate code. The justifications for using an intermediate layer essentially falls into two categories: encapsulation and distribution.

Encapsulation is the idea that software objects should be able to invoke services on other objects without knowing or caring about how those services are implemented. Encapsulation can reduce the amount of coding logic and make systems more maintainable. E.g., rather than having IF-THEN statements that determine which subroutine or function to call a developer can just send a message to the object and the object will select the appropriate code based on its type.

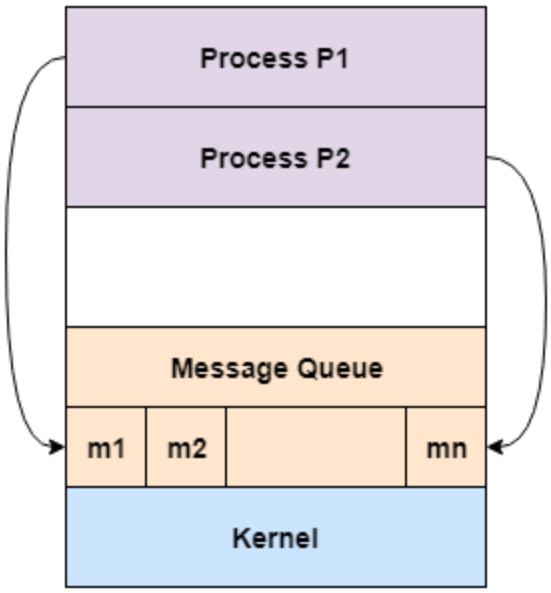
One of the first examples of how this can be used was in the domain of computer graphics. There are various complexities involved in manipulating graphic objects. For example, simply using the right formula to compute the area of an enclosed shape will vary depending on if the shape is a triangle, rectangle, ellipse, or circle. In traditional computer programming this would result in long IF-THEN statements testing what sort of object the shape was and calling the appropriate code. The object-oriented way to handle this is to define a class called Shape with subclasses such as Rectangle and Ellipse (which in turn have subclasses Square and Circle) and then to simply send a message to any Shape asking it to compute its area. Each Shape object will then invoke the subclass's method with the formula appropriate for that kind of object.

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message passing model of process communication is given as follows:

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message passing model of process communication is given as follows:



Message Passing Model

In the above diagram, both the processes P1 and P2 can access the message queue and store and retrieve data.

An advantage of message passing model is that it is easier to build parallel hardware. This is because message passing model is quite tolerant of higher communication latencies. It is also much easier to implement than the shared memory model.

However, the message passing model has slower communication than the shared memory model because the connection setup takes time.

Synchronous message passing

Synchronous message passing occurs between objects that are running at the same time. It is used by object-oriented programming languages such as Java and Smalltalk.

Synchronous messaging is analogous to a synchronous function call; just as the function caller waits until the function completes, the sending process waits until the receiving process completes. This can make synchronous communication unworkable for some applications. For example, large, distributed systems may not perform well enough to be usable. Such large, distributed systems may need to operate while some of their subsystems are down for maintenance, etc.

Asynchronous message passing

With asynchronous message passing the receiving object can be down or busy when the requesting object sends the message. Continuing the function call analogy, it is like a function call that returns immediately, without waiting for the called function to complete. Messages are sent to a queue where they are stored until the receiving process requests them. The receiving process processes its messages and sends results to a queue for pickup by the original process (or some designated next process).

Asynchronous messaging requires additional capabilities for storing and retransmitting data for systems that may not run concurrently, and are generally handled by an intermediary level of software (often called middleware); a common type being Message-oriented middleware (MOM).

MESSAGE PASSING VS. DISTRIBUTED SHARED MEMORY

Message passing	Distributed shared memory
Variables have to be marshalled	Variables are shared directly
Cost of communication is obvious	Cost of communication is invisible
Processes are protected by having private address space	Processes could cause error by altering data
Processes should execute at the same time	Executing the processes may happen with non-overlapping lifetimes

Parallel Computer Architectures

Switched Network Topologies
Processor arrays
Multiprocessors
Multicomputers

Switched Network Topologies

Binary Tree Network Topology In a binary tree network, the $2k-1$ nodes are arranged in a complete binary tree of depth $k-1$, as in Figure 2.1. The depth of a binary tree is the length of a path from the root to a leaf node. Each interior node is connected to two children, and each node other than the root is connected to its parent. Thus the degree is 3. The diameter of a binary tree network with $2k-1$ nodes is $2(k-1)$, because the longest path in the tree is any path from a leaf node that must go up to the root of the tree and then down to a dierent leaf node. If we let $n = 2k-1$ then $2(k-1)$ is approximately $2 \log_2 n$; i.e., the diameter of a binary tree network with n nodes is a logarithmic function of network size, which is very low. The bisection width is low, which means it is poor. It is possible to split the tree into two sets diering by at most one node in size by deleting either edge incident to the root; the bisection width is 1. As discussed above, maximum edge length is an increasing function of the number of nodes.

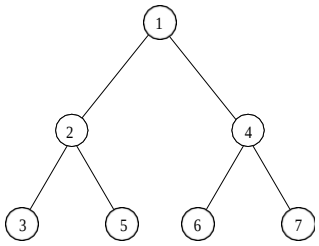
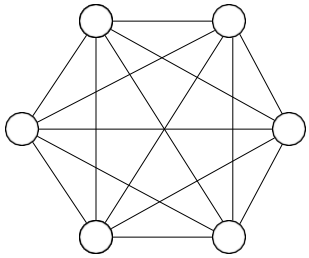


Figure : Binary tree topology with 7 nodes.

Fully-Connected Network Topology In a fully-connected network, every node is connected to every other node, as in Figure 2.2. If there are n nodes, there will be n(n − 1)/2 edges. Suppose n is even. Then there are n/2 even numbered nodes and n/2 odd numbered nodes. If we remove every edge that connects an even node to an odd node, then the even nodes will form a fully-connected network and so will the odd nodes, but the two sets will be disjoint. There are (n/2) edges from each even node to every odd node, so there are (n/2)2 edges that connect these two sets. Not removing any one of them fails to disconnect the two sets, so this is the minimum number. Therefore, the bisection width is (n/2)2 . The diameter is 1, since there is a direct link from any node to every other node. The degree is proportional to n, so this network does not scale well. Lastly, the maximum edge length will increase as the network grows, because nodes are not arbitrarily small.

Figure : Fully-connected network with 6 nodes.



Mesh Network Topology

In a mesh network, nodes are arranged in a q-dimensional lattice. A 2-dimensional lattice with 36 nodes is illustrated in Figure 2.3. The mesh in that gure is square. Unless stated otherwise, meshes are usually square. In general, there are k^2 nodes in a 2-dimensional mesh. A 3-dimensional mesh is the logical extension of a 2-dimensional one. It is not hard to imagine a 3-dimensional mesh. It consists of the lattice points in a 3-dimensional grid, with edges connecting adjacent points. A 3-dimensional mesh, assuming the same number of nodes in all dimensions, must have k^3 nodes. While we cannot visually depict q-dimensional mesh networks when q > 3, we can describe their properties. A q-dimensional mesh network has k^q nodes. k is the number of nodes in a single dimension of the mesh. Henceforth we let q denote the dimension of the mesh.

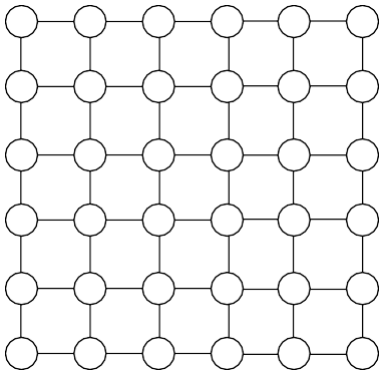


Figure : A two-dimensional square mesh with 36 nodes.

the 2-dimensional version of which is illustrated in Figure 2.4, is an extension of a mesh by the inclusion of edges between the exterior nodes in each row and those in each column. In higher dimensions, it includes edges between the exterior nodes in each dimension. It is called a torus because the surface that would be formed if it were wrapped around the nodes and edges with a thin

lm would be a mathematical torus, i.e., a doughnut. A torus, or toroidal mesh, has lower diameter than a non-toroidal mesh, by a factor of 2.

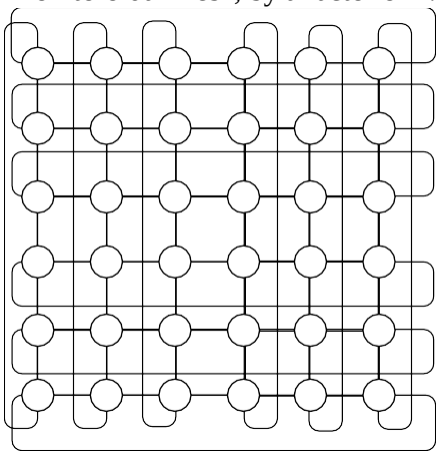
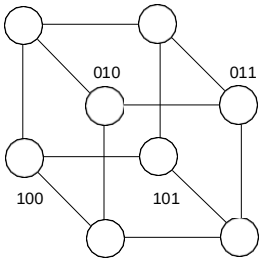
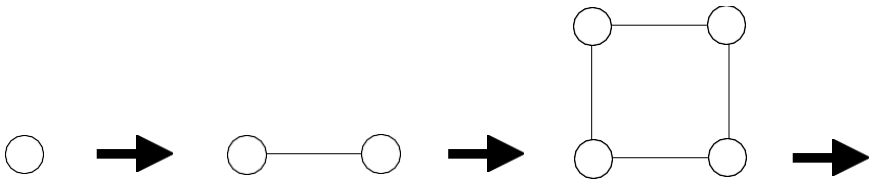


Figure : Two-dimensional mesh with toroidal connections.

10 11





Hypercube (Binary n-Cube)

A binary n-cube or hypercube network is a network with 2^n nodes arranged as the vertices of a *n*-dimensional cube. A hypercube is simply a generalization of an ordinary cube, the three-dimensional shape which you know. Although you probably think of a cube as a rectangular prism whose edges are all equal length, that is not the only way to think about it.

The node labels will play an important role in our understanding of the hypercube. Observe that

- The labels of two nodes di er by exactly one bit change if and only if they are connected by an edge.
- In an *n*-dimensional hypercube, each node label is represented by *n* bits. Each of these bits can be

  inverted (0->1 or 1->0), implying that each node has exactly *n* incident edges. In the 4D hypercube, for example, each node has 4 neighbors. Thus the degree of an *n*-cube is *n*.
- The diameter of an *n*-dimensional hypercube is *n*. To see this, observe that a given integer represented with *n* bits can be transformed to any other *n*-bit integer by changing at most *n* bits, one bit at a time. This corresponds to a walk across *n* edges in a hypercube from the rst to the second label.
- The bisection width of an *n*-dimensional hypercube is 2^{n-1}. One way to see this is to realize that all nodes can be thought of as lying in one of two planes: pick any bit position and call it b. The nodes whose b-bit = 0 are in one plane, and those whose b-bit = 1 are in the other. To split the network into two sets of nodes, one in each plane, one has to delete the edges connecting the two planes. Every node in the 0-plane is attached to exactly one node in the 1-plane by one edge. There are 2^{n-1} such pairs of nodes, and hence 2^{n-1} edges. No smaller set of edges can be cut to split the node set.
- The number of edges in an *n*-dimensional hypercube is $n\ 2^{n-1}$. To see this, note that it is true when $n = 0$, as there are 0 edges in the 0-cube. Assume it is true for all $k < n$. A hypercube of dimension *n* consists of two hypercubes of dimension *n*–1 with one edge between each pair of corresponding nodes in the two smaller hypercubes. There are 2^{n-1} such edges. Thus, using the inductive hypothesis, the hypercube of dimension *n* has $2 \cdot (n-1) \cdot 2^{n-2} + 2^{n-1} = (n-1) \cdot 2^{n-1} + 2^{n-1} = (n-1+1) \cdot 2^{n-1} = n \cdot 2^{n-1}$ edges. By the axiom of induction, it is proved.

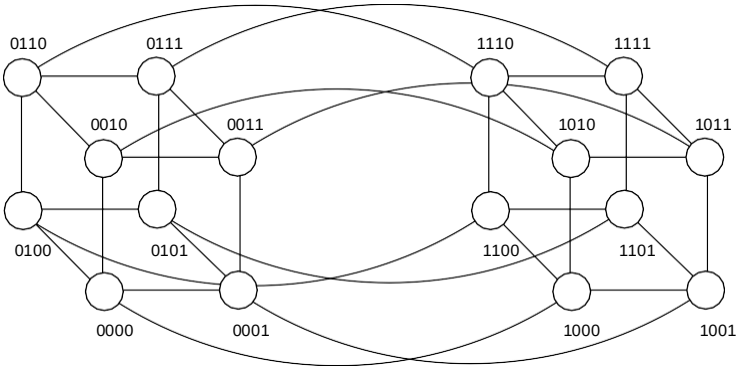


Figure : A 4-dimensional hypercube.

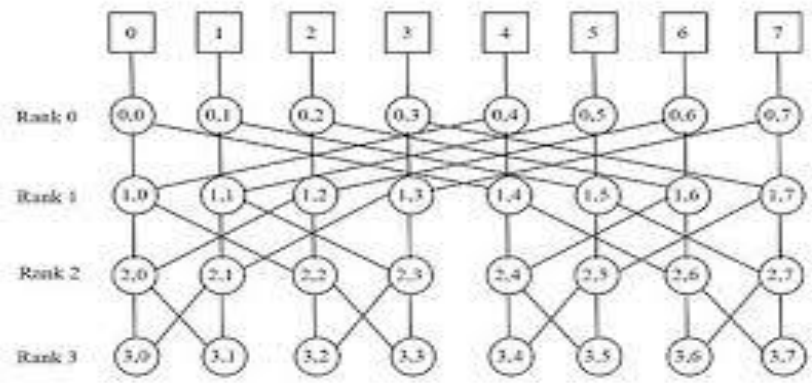
Butter y Network Topology

A butter y network topology consists of $(k + 1)2^k$ nodes arranged in $k + 1$ ranks, each containing $n = 2^k$ nodes. k is called the order of the network. The ranks are labeled 0 through k. Figure 2.7 depicts a butter y network of order 3, meaning that it has 4 ranks with $2^3 = 8$ nodes in each rank. The columns in the gure are labeled 0 through 7.

We describe two di erent methods for constructing a butter y network of order k.

Method 1:

- Create $k + 1$ ranks labeled 0 through k, each containing 2^k nodes, labeled 0 through $2^k - 1$.
- Let $[i, j]$ denote the node in rank i, column j.



For each rank, from 0 through k 1, connect all nodes $[i, j]$ to nodes $[i + 1, j]$. In other words, draw the straight lines down the columns as shown in Figure 2.7.

- For each rank, from 0 through k 1, connect each node $[i, j]$ to node $[i + 1, (j + 2^{k-i-1}$. This creates the diagonal edges that form the butter y pattern. For example, if $k = 3$, in rank 0, the node in column j is connected to the node in rank 1 in column $(j + 2^{k-1})\%2^k = (j + 4)\%8$, so the nodes 0,1,2, and 3 in rank 0 are connected to the nodes 4,5,6, and 7 respectively in rank 1, and nodes 4,5,6, and 7 in rank 0 are connected to nodes 0,1,2, and 3 respectively in rank 1.

Processor arrays

Multiple processors that carry out the same instruction in a given instruction cycle.

Typically paired with a "front-end" processor that is handing all the other work of the system.

Often called vector processors since operations on vectors typically involve identical operations on different data.

Processors have local memory and some form of interconnection network (often a 2-D mesh) allowing for communication between processors.

Masks can be used to selective enable/disable operations on individual processors. (e.g. think absolute value—only want to change sign if value is less than zero).

Processor array disadvantages

As late as the mid 2000's processor arrays were viewed as old technology. Reasons for this at the time included:

- many problems do not map well to strictly data-parallel solution conditionally executed parallel code is does not perform well
- not well suited for multi-user situations—requires dedicated access to processor array for best performance.
- cost does not scale-down well
- hard or impossible to do with COTS (commodity off-the-shelf) parts CPUs are relatively cheap (and getting faster)

Multiprocessors

- A computer with multiple CPUs or one or more multicore CPUs with shared memory.
- Common examples include dual-processor workstations and systems with multiple cores in a single processor. Past supercomputers include Cray X-MP and Y-MP.
- In a **Symmetric multiprocessor (SMP)** all processors are the same and have equal (but shared) access to resources.

- This is currently the standard model of desktop or laptop computers; a CPU with multiple cores, various levels of cache, and common access to single pool of RAM.

Multiprocessor cache issues

- Typically each CPU (or core; henceforth we'll just say CPU) has at least one level of cache. Data in the cache should be consistent with corresponding data in memory. When a CPU writes to its cache, that update must also be carried out in memory **and** in the caches of other CPUs where the updated data may also be stored. This is called the **cache coherence problem**.
- **Snooping** – each CPU's cache controller monitors bus so it is aware what is stored in other caches. System uses this information to avoid coherence problems.
- One solution is the **write invalidate protocol**: When one CPU write to its cache, the corresponding data in other's cache is marked as invalid. This causes a cache miss when any other CPU tries to read the data from its own cache.

Multiprocessor synchronization

- **Barriers** – Points in a parallel program where all processors must be at the same spot before proceeding.
- **Mutual Exclusion** – Often there are *critical sections* in code where the program must guarantee that only a single process accesses certain memory for a period of time. (e.g., don't want one process trying to read a memory location at the same instant another is writing to it.) **Semaphores** provide one way to do this.

Multiprocessor Memory Access

- **UMA (Unified Memory Access)**: Every processor has the same view and access to memory. Connections can be through a bus or a switched network. SMP machines typically belongs to the UMA category.
- Designs using a bus become less practical as the number of processors increases.
- **NUMA (NonUniform Memory Access) Distributed Multiprocessor**: Each processor has access to all memory but some access is indirect. Often memory is distributed and associated with each processor. There is a uniformly accessed virtual memory composed of all the distributed segments.

Multicomputers

- Commonly called *distributed memory computers*, these have multiple CPUs each having exclusive access to a certain segment of memory.
- Multicomputers can be **symmetrical** or **asymmetrical**.

- Symmetrical multicomputers are typically a collection of identical compute nodes.

Asymmetric vs symmetric multicomputers

- An asymmetrical multicomputer is a cluster of nodes with differentiated tasks and resources.
- Usually composed of front-end computer(s) called *head nodes* or *login nodes* and multiple back-end *compute nodes*.

- The head node typically handles program launching and supervision, I/O, network connections, as well as user-interface activities such as software development.
- Often programmed using either SPMD or MPMD model. In the SPMD case the program usually detects if it is running on the head node or a compute node and behaves accordingly.
- Supercomputer clusters that have dominated the supercomputer industries for the last 15-20 years are examples of asymmetrical multicomputers; many compute nodes and relatively few nodes for I/O, cluster supervision, and program development.
- Many small Beowulf clusters are symmetric multicomputers (or very nearly so).

General-purpose computing on graphics processing units (**GPGPU**, rarely **GPGP**) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple video cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing. In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to the specialization in each chip

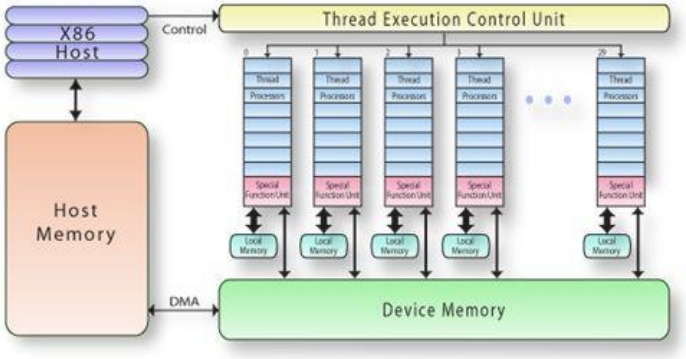
Essentially, a GPGPU pipeline is a kind of parallel processing between one or more GPUs and CPUs that analyzes data as if it were in image or other graphic form. While GPUs operate at lower frequencies, they typically have many times the number of cores. Thus, GPUs can process far more pictures and graphical data per second than a traditional CPU. Migrating data into graphical form and then using the GPU to scan and analyze it can create a large speedup.

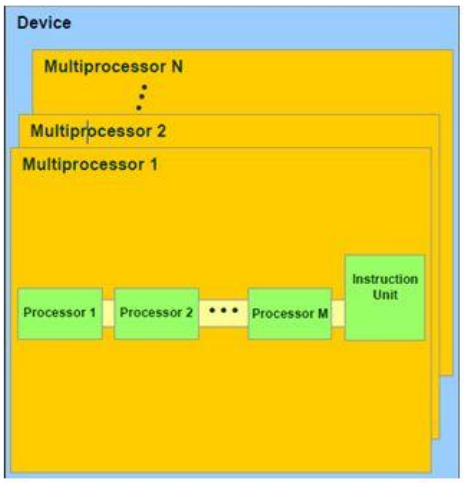
GPGPU pipelines were developed at the beginning of the 21st century for graphics processing (e.g., for better shaders). These pipelines were found to fit scientific computing needs well, and have since been developed in this direction.

A **graphics processing unit** (**GPU**) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics and image processing. Their highly parallel structure makes them more efficient than general-purpose central processing units (CPUs) for algorithms that process large blocks of data in parallel. In a personal computer, a GPU can be present on a video card or embedded on the motherboard. In certain CPUs, they are embedded on the CPU die.[1]

The term "GPU" was coined by Sony in reference to the PlayStation console's Toshiba-designed Sony GPU in 1994.[2] The term was popularized by Nvidia in 1999, who marketed the GeForce 256 as "the world's first GPU".[3] It was presented as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines".[4] Rival ATI Technologies coined the term "**visual processing unit**" or **VPU** with the release of the Radeon 9700 in 2002.[5]

Architecture of GPU





POSIX Threads, usually referred to as **pthreads**, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a *thread*, and creation and control over these flows is achieved by making calls to the POSIX Threads API. POSIX Threads is an API defined by the standard *POSIX.1c, Threads extensions*

Implementations of the API are available on many Unix-like POSIX-conformant operating systems such as FreeBSD, NetBSD, OpenBSD, Linux, macOS, Android[1], Solaris, Redox, and AUTOSAR Adaptive, typically bundled as a library **libpthread**. DR-DOS and Microsoft Windows implementations also exist: within the SFU/SUA subsystem which provides a native implementation of a number of POSIX APIs, and also within third-party packages such as *pthreads-w32*,[2] which implements **pthreads** on top of existing Windows API.

pthreads defines a set of C programming language types, functions and constants. It is implemented with a `pthread.h` header and a thread library.

There are around 100 threads procedures, all prefixed `pthread_` and they can be categorized into four groups:

- Thread management - creating, joining threads etc.
- Mutexes
- Condition variables
- Synchronization between threads using read/write locks and barriers

The POSIX semaphore API works with POSIX threads but is not part of threads standard, having been defined in the *POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993)* standard. Consequently, the semaphore procedures are prefixed by `sem_` instead of `pthread_`.

Example

An example illustrating the use of **pthreads** in C:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 5

void *perform_work(void *arguments){
  int index = *((int *)arguments);
  int sleep_time = 1 + rand() % NUM_THREADS;
  printf("THREAD %d: Started.\n", index);
  printf("THREAD %d: Will be sleeping for %d seconds.\n", index, sleep_time);
  sleep(sleep_time);
  printf("THREAD %d: Ended.\n", index);

}

int main(void) {
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int i;
  int result_code;

  //create all threads one by one
  for (i = 0; i < NUM_THREADS; i++) {
    printf("IN MAIN: Creating thread %d.\n", i);
    thread_args[i] = i;
    result_code = pthread_create(&threads[i], NULL, perform_work, &thread_args[i]);
    assert(!result_code);
  }

  printf("IN MAIN: All threads are created.\n");

  //wait for each thread to complete
  for (i = 0; i < NUM_THREADS; i++) {
    result_code = pthread_join(threads[i], NULL);
    assert(!result_code);
    printf("IN MAIN: Thread %d has ended.\n", i);
  }

  printf("MAIN program has ended.\n");
  return 0;
}
```

This program creates five threads, each executing the function *perform_work* that prints the unique number of this thread to standard output. If a programmer wanted the threads to communicate with each other, this would require defining a variable outside of the scope of any of the functions, making it a global variable. This program can be compiled using the gcc compiler with the following command:

```
gcc pthreads_demo.c -lpthread -o pthreads_demo
```

Here is one of the many possible outputs from running this program.

```
IN MAIN: Creating thread 0.
IN MAIN: Creating thread 1.
IN MAIN: Creating thread 2.
IN MAIN: Creating thread 3.
THREAD 0: Started.
IN MAIN: Creating thread 4.
THREAD 3: Started.
THREAD 2: Started.
THREAD 0: Will be sleeping for 3 seconds.
THREAD 1: Started.
THREAD 1: Will be sleeping for 5 seconds.
THREAD 2: Will be sleeping for 4 seconds.
THREAD 4: Started.
THREAD 4: Will be sleeping for 1 seconds.
IN MAIN: All threads are created.
THREAD 3: Will be sleeping for 4 seconds.
THREAD 4: Ended.
THREAD 0: Ended.
IN MAIN: Thread 0 has ended.
THREAD 2: Ended.
THREAD 3: Ended.
THREAD 1: Ended.
IN MAIN: Thread 1 has ended.
IN MAIN: Thread 2 has ended.
IN MAIN: Thread 3 has ended.
IN MAIN: Thread 4 has ended.
MAIN program has ended.
```

Alternative to locking – STM Systems Due to these difficulties with locks, alternative approaches were looked into. Software transactional memory (STM) is an approach which has garnered significant interest as an elegant alternative for developing parallel programs. Software transactions are units of execution in memory which enable concurrent threads to execute seamlessly. Software transactions address many of the shortcomings of lock based systems. This idea originated from transactions in databases.

Unlike the lock-based programming, STM approach is optimistic: a thread completes modifications to shared memory without regard for what other threads might be doing, recording every read and write that it is performing in a log. The STM system then looks into the log and validates if the actions of the thread can be allowed to become permanent (called committed) or not (aborted then). The benefit of this approach is increased concurrency: no thread needs to wait for access to a resource, and different threads can safely and simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock. Another advantages of STMs is that it provides a very promising approach for composing software components .

STMs achieve composition through nesting of transactions. A transaction is called nested if it invokes another transaction as a part of its execution. Programming support for STMs In order to execute code accessing shared data-items as transactions, the user designates piece of code in an atomic block. For instance, the following code will be executed as a transaction by the STM system. atomic { if (x != null) x.foo(); y = true; } The STM system's responsibility is to execute these transactions as if they were atomic — as if the entire body of the transaction were executed at a single moment of time. As discussed above, a transaction executes to completion then it is committed and its effects are visible to other transactions. Otherwise it is aborted and none of its effects are visible to other transactions. Thus it can be seen that not much change is required to the existing code. Sections of the

code which access shared objects have to be designated as transactions. The STM system ensures that they execute atomically. Conclusion Software transactional memory (STM) is an approach which has garnered significant interest as an elegant alternative for developing parallel programs. Software transactions are units of execution in memory which enable concurrent threads to execute seamlessly. Software transactions can be employed to address many of the shortcomings of lock based systems.